Tabelle hash a indirizzamento chiuso e a indirizzamento aperto

Loris Rossi

21 maggio 2017

Introduzione

Una tabella hash è una struttura dati che mette in corrispondenza una data chiave con un dato valore. Essa utilizza una *funzione hash* per calcolare l'indice di un array di slot in cui è possibile trovare il valore desiderato.

Idealmente la funzione hash dovrebbe assegnare ad ogni chiave un unico slot, ma poichè molto spesso l'universo delle chiavi è maggiore del numero di slot, alcune chiavi saranno associate allo stesso slot. Questo evento si chiama *collisione*, e si possono distinguere due gruppi di tabelle hash in base a come vengono gestite le collisioni:

- Tabelle hash a indirizzamento chiuso: tutti gli elementi che sono associati allo stesso slot vengono posti in una lista concatenata.
- Tabelle hash a indirizzamento aperto: tutti gli elementi sono memorizzati nella stessa tabella hash, e quando si incontra una collisione si calcola l'indice di un altro slot, mediante una scansione.

La caratteristica principale di una tabella hash è che, sotto opportune condizioni, le operazioni fondamentali sulla tabella richiedono in media un tempo $O(1)$.

Perchè ciò si verifichi, è necessario che la funzione di hash distribuisca uniformemente i valori in ogni slot. Se qualsiasi elemento ha la stessa probabilità di essere associato a uno qualsiasi slot, si parla di *hashing uniforme semplice*.

Inoltre si definisce *fattore di carico* α della tabella hash il rapporto n/k, dove n è il numero di valori presenti nella tabella, e k è il numero di slot. Il tempo atteso costante assume che il fattore di carico sia sotto una certa soglia. Infatti, se il fattore di carico cresce, le operazioni sulla tabella hash rallentano.

Contesto

L'obiettivo di questo progetto è confrontare le prestazioni di una tabella hash a indirizzamento chiuso e una a indirizzamento aperto in termini di tempo di esecuzione, in diverse casistiche. A tale scopo, viene utilizzato un file contenente 6782 nomi diversi, utilizzati come chiavi. Come valori si sono generati in modo casuale dei numeri interi, che nel codice sono stati chiamati *account id* per simulare un caso reale di applicazione.

Il codice inserisce tutte le coppie [*chiave, valore*] nella tabella hash, e poi vengono ricercate tutte, calcolando quindi il tempo di inserimento e il tempo di ricerca di tutti gli elementi.

Per ridurre gli errori casuali, la procedura viene eseguita per 1000 volte, e infine viene calcolata la media dei tempi registrati.

Come funzione hash è stato scelto il metodo della divisione. Dato che le chiavi sono delle stringhe, ad ogni carattere della stringa viene associato il corrispondente valore numerico ASCII, e viene eseguito il modulo della somma di questi valori numerici.

Si sono provate diverse configurazioni, in cui si sono variati il fattore di carico e il tipo di scansione.

In particolare, per l'indirizzamento chiuso sono state usate le seguenti dimensioni: 2 ($\alpha = 3991$), 100 ($\alpha = 67.82$), 1000 ($\alpha = 6.782$), 7000 ($\alpha = 0.969$), 20000 ($\alpha = 0.339$). Il caso in cui la dimensione è 2 simula il caso in cui la funzione hash è fortemente sbilanciata, infatti vengono riempite solo 2 liste, e le operazioni sulla tabella rallentano notevolmente rispetto agli altri casi.

Per l'indirizzamento aperto, invece, sono state usate la scansione lineare e il doppio hashing, e le seguenti dimensioni: 6907 ($\alpha = 0.982$), 9929 ($\alpha = 0.683$), 14979 ($\alpha = 0.458$), 20117 ($\alpha = 0.337$).
(in questo caso le dimensioni corrispondono a dei numeri primi, in modo da ottenere un funzionamento corretto del doppio hashing)

Risultati

Qui sotto sono riportati i risultati ottenuti. Si ricorda che i tempi elencati sono i tempi medi calcolati su 1000 iterazioni, in cui si inseriscono e si ricercano tutti gli elementi.

Indirizzamento chiuso

Fattore di carico	Tempo di inserimento	Tempo di ricerca
0.339	2.09 ms	1.82 ms
0.969	2.09 ms	1.79 ms
6.782	2.12 ms	1.84 ms
67.82	6.26 ms	5.77 ms
3391	118.24 ms	116.13 ms

Indirizzamento aperto, doppio hashing

Fattore di carico	Tempo di inserimento	Tempo di ricerca
0.337	18.98 ms	12.04 ms
0.458	19.97 ms	12.22 ms
0.683	21.17 ms	12.93 ms
0.982	31.46 ms	19.36 ms

Indirizzamento aperto, scansione lineare

Fattore di carico	Tempo di inserimento	Tempo di ricerca
0.337	1455 ms	871 ms
0.458	1426 ms	857 ms
0.683	1527 ms	885 ms
0.982	1481 ms	892 ms

Come è facile notare, la tabella hash a indirizzamento chiuso ha prestazioni migliori della tabella hash a indirizzamento aperto in quasi tutti i casi.
Il caso in cui ci sono solo 2 slot ha prestazioni notevolmente peggiori (quasi due ordini di grandezza) rispetto agli altri casi, come ci si aspettava.
La tabella hash ad indirizzamento aperto con scansione lineare risulta molto lenta probabilmente perchè la funzione hash utilizzata non è uniforme, quindi ci sono tante collisioni vicine che rallentano l'esecuzione.

Struttura del codice

Il codice si articola in 3 file:

- hash-table-lib.c
- chained-hash-table.c
- open-hash-table.c

In *hash-table-lib.c* sono presenti i metodi di base per gestire i nodi e le liste. Il nodo utilizzato, *DataSlot*, è una struttura che contiene una stringa *key*, un numero intero *value*, e l'indirizzo di memoria del nodo successivo *link*.
Inoltre sono presenti anche le funzioni *GetNames*, che legge i nomi dal file names.csv e li salva in un array di stringhe, e *GetTimeMs64*, che restituisce il tempo trascorso dal 1 gennaio 1970 in millisecondi, utilizzata per calcolare i tempi di esecuzione.

In *chained-hash-table.c* e in *open-hash-table.c* sono presenti le relative funzioni principali della tabella hash (funzione hash, scansione, stampa, inserimento, ricerca, delete). Per la tabella hash a indirizzamento aperto non è stata implementata l'operazione *delete* dato che la ricerca di un elemento termina se si incontra un nodo vuoto, e cancellare un elemento dalla tabella può alterare questo comportamento.
Entrambi i file hanno la stessa funzione *main*, in cui vengono generate le coppie [*chiave,valore*]; successivamente si entra in un ciclo for che ad ogni iterazione crea una tabella hash, inserisce tutti gli elementi, ricerca tutti gli elementi e salva il tempo di esecuzione totale di entrambe le operazioni. Al termine del ciclo vengono eseguite le medie dei tempi di esecuzione, che vengono stampate a video.

Non viene riportato il main di *open-hash-table.c* dato che è lo stesso di *chained-hash-table.c*.

hash-table-lib.c

```c
#include <stdio.h> // printf
#include <stdlib.h> // malloc, free, realloc
#include <string.h> // strcmp
#include <inttypes.h> // uint64_t


// DataSlot è un nodo di una lista, che contiene la chiave 'key' e il valore 'value'
struct SDataSlot {
  char* key;
  int value;
  struct DataSlot* link;
};
typedef struct SDataSlot DataSlot;
typedef DataSlot* List;

DataSlot* DataSlotCreate(char* key, int value) {
  DataSlot *new;
  new = (DataSlot *) malloc(sizeof(DataSlot));
  if (new == NULL)
    return NULL;
  new->key = key;
  new->value = value;
  new->link = NULL;
  return new;
}

void DataSlotPrint(DataSlot* data_slot) {
  if (data_slot != NULL)
    printf("Key: %s, Value: %d\n", data_slot->key, data_slot->value);
  else
    printf("Slot vuoto.\n");
}

void DataSlotDestroy(DataSlot* data_slot) {
  free(data_slot);
}

List ListCreate() {
  return NULL;
}

void ListPrint(List list) {
  DataSlot* current = list;
  while (current != NULL) {
    DataSlotPrint(current);
```

```
46      current = current->link;
47    }
48  }
49
50  DataSlot* ListSearch(List list, char* key) {
51    DataSlot* current = list;
52    while (current != NULL) {
53      if (!strcmp(current->key, key))
54        return current;
55      current = current->link;
56    }
57    return NULL;
58  }
59
60  // Inserimento in testa
61  // Se la chiave è già presente, non viene inserito il nuovo
        elemento
62  List ListInsert(List list, char* key, int value) {
63    DataSlot* isPresent = ListSearch(list, key);
64    if (isPresent != NULL) {
65      printf("Chiave già presente, inserimento di [%s, %d] fallito
        .\n", key, value);
66      return list;
67    }
68    else {
69      DataSlot* new = DataSlotCreate(key, value);
70      if (new == NULL) {
71        printf("Errore di allocazione di memoria");
72        exit(1);
73      }
74      new->link = list;
75      return new;
76    }
77  }
78
79  List ListDelete(List list, char* key) {
80    DataSlot* current = list;
81    DataSlot* previous = NULL;
82    // Ricorda che 'strcmp' vale 0 se le stringhe sono uguali
83    while ((current != NULL) && strcmp(current->key, key)) {
84      previous = current;
85      current = current->link;
86    }
87    if (current != NULL) {
88      // Slot trovato
89      if (previous == NULL)
90        list = current->link;
91      else
92        previous->link = current->link;
```

```
93
94      DataSlotDestroy(current);
95    }
96    return list;
97  }
98
99  void GetNames(char* filename, char** array_with_names, int
        kRowsNumber) {
100
101   FILE* fp;
102   char buffer[255];
103
104   fp = fopen(filename, "r");
105   if (fp == NULL) {
106     printf("Errore, il file '%s' non esiste.\n", filename);
107     exit(1);
108   }
109
110   int i=0;
111   while (fgets(buffer, 255, (FILE*) fp) && i < kRowsNumber) {
112     buffer[strcspn(buffer, "\r\n")] = 0; // rimuove il new line
        dal buffer
113     strcpy(array_with_names[i], buffer);
114     i++;
115   }
116 }
117
118 // Restituisce il tempo trascorso dal 01/01/1970 in millisecondi
        .
119 // Nota bene: questa funzione funziona solo su sistemi Linux.
120 uint64_t GetTimeMs64() {
121  struct timeval tv;
122
123  gettimeofday(&tv, NULL);
124
125  uint64_t ret = tv.tv_usec;
126  // Converte i microsecondi (10^-6) in millisecondi (10^-3)
127  ret /= 1000;
128
129  // Aggiunge i secondi (10^0) dopo averli convertiti in
        millisecondi (10^-3)
130  ret += (tv.tv_sec * 1000);
131
132  return ret;
133 }
```

chained-hash-table.c

```
/**
*   Implementazione di una tabella hash a indirizzamento chiuso, con gestione
*   delle collisioni mediante liste concatenate.
*   Le chiavi sono delle stringhe, e i valori associati sono dei numeri interi.
*   L'hashing viene effettuato mediante il metodo della divisione.
*/

#include "hash-table-lib.c"
#include <time.h>

const int kHashTableSize = 7000;

// Hashing con metodo della divisione
int HashCode(char* string) {
  int sum = 0;
  for (int i = 0; string[i] != '\0'; i++)
    sum += string[i]*100;
  return sum % kHashTableSize;
}

void HashTablePrint(List* hash_table) {
  printf("STAMPA HASH TABLE\n");
  for (int i=0; i<kHashTableSize; i++) {
    if (hash_table[i] != NULL) {
      printf("Slot corrente: %d\n", i);
      ListPrint(hash_table[i]);
      printf("\n");
    }
  }
}

void HashTableInsert(List* hash_table, char* key, int value) {
  int x = HashCode(key);
  hash_table[x] = ListInsert(hash_table[x], key, value);
}

DataSlot* HashTableSearch (List* hash_table, char* key) {
  int x = HashCode(key);
  return ListSearch(hash_table[x], key);
}

void HashTableDelete (List* hash_table, char* key) {
  int x = HashCode(key);
  hash_table[x] = ListDelete(hash_table[x], key);
```

```
}


int main() {
  const int kRowsNumber = 6782; // Numero di righe del file csv
  char* names[kRowsNumber];
  int account_id[kRowsNumber];
  uint64_t t0, t1; // Utilizzate per calcolare i tempi di
    esecuzione
  uint64_t insert_time=0, search_time=0; // Tempo totale di
    inserimento e ricerca
  DataSlot* item; // Variabile d'appoggio per la ricerca
  const int kIterations = 1000; // Numero iterazioni del ciclo
    for principale

  // Alloca la memoria per le stringhe dei nomi
  for (int i=0; i<kRowsNumber; i++)
  names[i] = (char*) malloc(255);

  // Legge i nomi dal file csv e li salva nell'array names[]
  GetNames("names.csv", names, kRowsNumber);

  // Assegna un numero casuale agli account_id
  srand(time(NULL));
  for (int i=0; i<kRowsNumber; i++)
    account_id[i] = rand();

  printf("TABELLA HASH AD INDIRIZZAMENTO CHIUSO\n");
  for (int j=0; j<kIterations; j++) {
    printf("Iterazione: %d\n", j+1);

    List hash_table[kHashTableSize];

    // Inizializza gli slot della tabella hash
    for (int i=0; i<kHashTableSize; i++)
    hash_table[i] = NULL;


    // Inserimento
    t0 = GetTimeMs64();
    for (int i=0; i<kRowsNumber; i++)
    HashTableInsert(hash_table, names[i], account_id[i]);
    t1 = GetTimeMs64();
    insert_time += (t1-t0);

    // Ricerca
    t0 = GetTimeMs64();
    for (int i=0; i<kRowsNumber; i++)
    item = HashTableSearch(hash_table, names[i]);
```

```
91      t1 = GetTimeMs64();
92      search_time += (t1-t0);
93    }
94
95    printf("INSERIMENTO, Tempo medio: %.2f millisecondi\n", (float
        )insert_time/kIterations);
96    printf("RICERCA, Tempo medio: %.2f millisecondi\n", (float)
        search_time/kIterations);
97
98    return 0;
99  }
```

open-hash-table.c

```
/**
*   Implementazione di una tabella hash a indirizzamento aperto.
*   Le chiavi sono delle stringhe, e i valori associati sono dei
     numeri interi.
*   Sono implementate sia la scansione lineare che il doppio
    hashing.
*/

#include "hash-table-lib.c"
#include <time.h>

const int kHashTableSize = 7000;

// Metodo ausiliare di HashCode()
// Hashing con metodo della divisione.
int HashAux(char* string) {
  int sum = 0;
  for (int i = 0; string[i] != '\0'; i++)
    sum += string[i];
  return sum % kHashTableSize;
}

// Metodo ausiliare di HashCode()
// Hashing con metodo della divisione.
int HashAux2(char* string) {
  int sum = 0;
  for (int i = 0; string[i] != '\0'; i++)
    sum += string[i];
  return sum % (kHashTableSize - 1);
}

// Scansione lineare
int HashCodeLinear(char* string, int i) {
  return (HashAux(string) + i) % kHashTableSize;
}

// Doppio hashing
int HashCode(char* string, int i) {
  return (HashAux(string) + i*HashAux2(string)) % kHashTableSize
    ;
}

void HashTablePrint(DataSlot* hash_table[]) {
  printf("STAMPA HASH TABLE\n");
  for (int i=0; i<kHashTableSize; i++) {
    if (hash_table[i] != NULL) {
      printf("Slot corrente: %d\n", i);
```

```
45        DataSlotPrint(hash_table[i]);
46        printf("\n");
47      }
48    }
49  }
50
51  DataSlot* HashTableSearch (DataSlot* hash_table[], char* key) {
52    int i = 0;
53    int hash = HashCode(key, i);
54    while (hash_table[hash] != NULL && i < kHashTableSize) {
55      if (!strcmp(hash_table[hash]->key, key))
56        return hash_table[hash];
57      i++;
58      hash = HashCode(key, i);
59    }
60    return NULL;
61  }
62
63  // Se la chiave è già presente, non viene inserito il nuovo
      elemento
64  void HashTableInsert(DataSlot* hash_table[], char* key, int
      value) {
65    DataSlot* is_present = HashTableSearch(hash_table, key);
66    if (is_present != NULL) {
67      printf("Chiave già presente, inserimento di [%s, %d] fallito
      .\n", key, value);
68      return;
69    }
70    int hash;
71    for (int i=0; i<kHashTableSize; i++) {
72      hash = HashCode(key, i);
73      if (hash_table[hash] == NULL) {
74        hash_table[hash] = DataSlotCreate(key, value);
75        return;
76      }
77    }
78    printf("Errore, tabella hash piena.\n");
79  }
```